APUNTES DE LA ASIGNATURA

PROGRAMACION

Año 2000
Profesor: Lorenzo Flores
Lenguaje de Programación: C
Autor: Jesús Arnáiz - Centro San Valero

- Nivel exigido:

Para seguir estos apuntes se requieren unos conocimientos del lenguaje C a nivel básico.

- Tipos de Datos:

enteros:

char 1 caracter (1 byte = 8 bits)

int El tamaño de un registro del micro -> depende de la arquitectura y del S.O. para arquitecturas i386 (Intel)
MSDOS -> 2 bytes (16 bits) en modo protegido 4 bytes.
LINUX, WIN32 -> 4 bytes (32 bits)

long 4 bytes o 32 bits.

decimales:

float 4 bytes (32 bits), para el exponente y la mantisa.

double 8 bytes (64 bits), también para exponente y mantisa.

- Modificadores a los tipos básicos de datos:

De signo:

Signed Con signo (normalmente por defecto)

Unsigned Sin signo

Ejemplo:

unsigned char

Tenemos 8 bits (1 byte) para trabajar datos sin signo es decir un rango de 0...255

signed int

(En MS-DOS p.ej.) Tenemos 16bits (2 bytes) para trabajar datos numéricos, teniendo en cuenta que el bit de mayor peso se utiliza para el signo es decir 1 negativo 0 positivo luego podemos obtener un rango de:

-32768...32767

De visibilidad:

s**tatic** Se inicializará la variable a 0 y una vez terminado el módulo permanece estática, es decir mantiene el valor a la siguiente llamada (se utiliza en funciones recursivas).

Ejemplo:

```
void Contador( void )
{
      static int conta;

      printf( "%d", conta++ );
}
```

Si llamamos a esta función sucesivas veces veremos como el valor de conta se va modificando con cada llamada, es decir permanece estático, aunque no por ello varía su visibilidad, vamos, que desde el main no podrás ver su valor ya que es una variable local.

a**uto** — **S**e crea y se destruye en el ámbito en el que se declara, y además no se inicializa (por defecto todas las variables que no se declaren como static)

extern — Sirve para declarar una referencia a una variable que puede estar en otro módulo. (Sólo se puede inicializar donde se declara)

Ejemplo:

fichero1.c

```
int mivar = 0;
...
```

fichero2.c

```
/* Referencia a la variable declarada en fichero1.c */
extern int mivar;
...
```

Modo de acceso:

const — Hace que una variable tome el valor con el que la inicialicemos el cual ya no podrá ser cambiado.

volatil — Indica que el valor de la variable puede cambiar incluso por condiciones eternas al programa que la declara. (Ejemplo una variable puntero al contador del reloj (Es lógico que queramos que el S.O. pueda seguir modificando su valor))

Los modificadores de visiblidad se pueden aplicar también a las definiciones de las funciones.

Ejemplo:

```
/*    Le indicamos al compilador que usaremos la función getch declarada
      en otro módulo */
extern int getch( void );
```

- La función main:

Es la función principal (como su nombre indica) del programa, en caso de tratarse de un programa por módulos (varios ficheros .c y .h) sólo puede existir una. Su declaración es así:

[tipo] [nom.] [param]

```
  void main ()
```

[tipo] Es el valor devuelto, si se omite se supone int.

[nom.] Nombre, el nombre de la función.

[param.] Parámetros que recibe la función si se deja vacío es igual a void.

Argumentos pasados al programa:

La declaración de main para trabajar con argumentos es la siguiente, y aunque se suele utilizar estos nombres para las variables, podemos usar los que más nos gusten. De los que se citan a continuación podemos usar 0, 1, 2 o 3.

```
main ( int argc, char ** argv, char **env )
```

argc Número de argumentos (mínimo 1 (el nombre del programa))

****argv** También se puede declarar *argv[] o argv[][]. Es una array bidimensional de chars es decir una tabla de caracteres o un vector de cadenas, contiene uno a uno los argumentos del programa

****env** Similar al anterior, sólo que en este caso se trata de las variables de entorno declaradas en el sistema.

- Instrucciones para el prepocesador:

Se declaran con una almhoadilla "#" en la primera columna, son las siguientes:

include
define
ifdef
ifndef
endif

Al generar un programa se ejecutan los siguientes programas:

***cpp (C Pre Procesor):**

Encargado de expandir el fuente (sustituye los #include por el fichero ...) generando uno nuevo.

***cc (C Compiler):**

Crea la tabla de símbolos y crea el módulo objeto (con referencias a funciones externas).

***link (Linkador) :**

Incluye las funciones externas a mi propio programa y lo junta todo generando un ejecutable.

Tener en cuenta que la mayoría de los compiladores permiten ejecutar sólo uno o varios de los pasos anteriores, por ejemplo en el GCC (GNU C Compiler) usando la opción -c le indicamos al compilador que pare antes de linkar (útil para compilar una librería que luego enlazarás con tu programa principal)

- Distintas formas de usar el modificador void:

void f();

Indica que la función f no devuelve ningún valor.

void *variable;
void *f();

Variable/Función de tipo puntero a un tipo de datos desconocido.

- printf:

(print formatted -> Imprimir formateado)

printf("formato", ...);

Su número de parámetros es variable, ya que en C las funciones pueden tener parámetros variables en tipo y en número.

El "formato" puede contener literales -> "Pepito", patrones "%c" y secuencias de escape "\t"

Algunos patrones:

%s (String) Cadena.
%c (Character) Caracter.
%d (Decimal) Valor entero decimal
%x Hexadecimal.
%f (Flotante).

Los patrones numéricos pueden estar modificados por una l que indica tipo long.

Si en el texto queremos escribir el % podemos usar un escape \% o duplicar el "%" %%.

Algunas secuencias de escape:

\n (New line) Retorno de carro y salto de línea.
\r Retorno de carro.
\t Tabulador (normalmente desde la columna 0 van de 8 en 8 o 9 en 9).
\a Pitido de la campana.
**\" o ** ... para escribir las comillas, la contrabarra y otros caracteres especiales.

– Bucles e iteraciones:

```
      [ini]  [cond] [inc]
for (      ;      ;     )
(...)
```

[ini] La inicialización de las variables.
[cond] Condición durante la cual se seguirá repitiendo el bucle.
[inc] Incrementos o pasos.

Se ejecuta el bucle mientras se cumpla la condición, ejecutándose cada vez el incremento.

```
      [cond]
while (     )
(...)
```

[cond] Condición durante la cual continua la ejecución del bucle
Se ejecuta si y sólo si se cumple la condición y mientras esta se cumpla.

```
do
(...)
       [cond]
while (     );
```

Se ejecuta una vez como mínimo y a partir de ahí mientras se cumpla la condición.

Instrucciones dentro de las iteraciones:

break: Sale del bucle en el mismo momento que se encuentra esta instrucción.

continue: obligas al programa a saltar al comienzo del bucle.

goto: Salto a una etiqueta (ya no se emplea)

- Control de Flujo:

Operadores que podemos emplear para las condiciones:

== Igual
!= Distinto
< Menor
> Mayor
<= Menor o Igual
>= Mayor o Igual
&& Condición 'Y'
|| Condición 'O'

En C se compara de izquierda a derecha, es decir:

(A > B)

Significa: *¿ A es mayor que B ?*

IF:

If (cond)
(...)
else
(...)
Se ejecuta si se cumple la condición impuesta en ***cond***, si así lo deseamos, podemos incluir el trozo que se ejecutará en caso de no cumplirse, añadiendo la línea else.

SWITCH:

```
switch ( cond )
{
      case ____
            (...)
            break;
      case ____
            (...)
            break;
      (...)
      default

}
```

El break no es obligatorio, ya que es posible que nos interese dar la posibilidad de que se cumplan varias, ya que en el switch de C el programa no termina las comparaciones hasta encontrar la llave del switch o un break.

default: Es la sentencia que se ejecutará por defecto.

- Tipos de Datos definidos por el usuario:

Estructuras (Registros):

```
struct [nombre]
{
      tipo var1;
      tipo var2;
      (...)
}
```

Para acceder a los datos de un registro se utiliza se hace con el operador punto '.', de la siguiente manera:

```
dato.var1 = 1;
```

En caso de que dato no sea una variable tipo estructura, sino un puntero a estructura se emplea el operador flecha '->':

```
dato->var1 = 1;
```

Ejemplo:

```
struct mireg
{
      char nombre[40];
      int telefono[10];
}

void main( void )
{
      struct mireg registro1;

      strcpy( registro1.nombre, "Mi nombre" );
      strcpy( registro1.telefono, "976387403" );
}
```

Campos de bits:

Son un tipo de estructura en el que algunos de sus miembros tiene características especiales.

```
struct [nombre]
{
      unsigned int var:n;
      (...)
}
```

siendo ***n*** el número de bits que queremos que contenga esa variable. Se pueden combinar con cualquier otro tipo de datos. (*NOTA: el tamaño que devolverá un sizeof echo a un tipo estructura campo de bits es el del valor siguiente (ya que sizeof siempre devuelve cantidad de bytes).*

Ejemplo:

```
struct pepito
{
      char letra;
      unsigned bits:2;
}
```

Aunque el uso normal que se le suele dar a un campo de bits es el de agrupar condiciones CIERTAS o FALSAS, cuando un valor combine un cierto tipo de situaciones o estados. (*Ejemplo:* estado del modem (CTS, RTS ...))

Uniones:

Se suelen emplear en conjunción con los campos de bits, éstas nos permiten contener una misma variable en dos zonas de memoria diferentes y cuyo tipo de dato no tiene por qué ser igual (por ejemplo contener el valor 100 como un char y como un campo de 8 bits.)

Ejemplo:

```
struct cbits
{
      unsigned bit1:1;
      unsigned bit2:2;
      (...8)
}

union tag
{
      char c;
      struct cbits b;
}

void main( void )
{
      union tag mivar;

      mivar.c = 0xFF
      printf( "Bit1: %d \n", mivar.b.bit1 );
}
```

– Definirnos un tipo propio:

Muchas veces, resulta más cómodo definirnos un tipo de dato que emplearemos igual que si se tratase de uno propio del C, a partir de nuestras estructuras o uniones, o para dar claridad al código.

Para ello empleamos *typedef,*

typedef:

typedef **[tipo] [nombre(s)]**

Ejemplo:

```
#define FALSO 0
#define CIERTO !FALSO

typedef unsigned char byte, boleano, mio;

void main(void)
{
      byte mivar;
      boleano continuar = FALSO;

      if ( continuar )
      (...)
}
```

Aunque, como he indicado también podremos usarlo con nuestras estructuras y uniones.

Ejemplo:

```
struct reg
{
      int dato1;
      char dato2[20];
}

typedef struct reg REG;
typedef struct reg *PREG;

{
      REG mireg;
      (...)
}
```

ó

```
typedef struct
{
      int dato1;
      char dato2[20];
} REG, *PREG;
```

- Punteros:

Como bien sabemos, en ellos reside gran parte de la potencia del lenguaje C, se trata de un tipo de variable que lo que contiene es una dirección de memoria en la que se supone se encuentra otra variable. Es decir apunta a algún sitio.

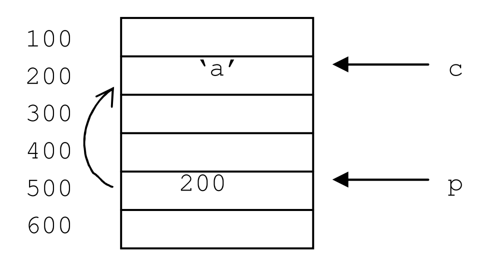
Se declaran de la siguiente manera:

[tipo al que apuntan] ***[nombre]**

```
char *p;
```

'p' es una variable que contendrá la dirección de memoria en la que se aloja un char;

dirección Contenido Variable



Lo que representa el esquema anterior, bien podría ser el siguiente caso:

```
{
      char c = 'a';
```

```
        char *p;

        p = &c;
}
```

Formas de acceder a un puntero:

- ***p** Obtenemos el valor de la variable a la que apunta, 'a'
- **p** La dirección de memoria dónde está la variable a la que apuntamos, 200.
- **&p** La dirección donde está alojado p, 500.

Aritmética de punteros y Arrays:

Tal y como hemos visto, un puntero no es más que una variable que apunta a otra, un uso corriente de éstos es para recorrer arrays, un array no es más que una serie de posiciones de memoria consecutivas referenciadas por una variable:

```
int numero;
int v[4] = { 1, 2, 3, 4 };

{
        /* Asignamos el cuarto elemento (4) a "numero" */
        numero = v[3];
}
```

Esto mismo podemos hacerlo con punteros gracias a la aritmética de punteros. A un puntero podemos decirle avanza n posiciones o retrocede n posiciones, y el sólo sabrá las posiciones de memoria que tendrá que avanzar dependiendo de el tipo al que apunte, es decir si apunta a un tipo byte "sabrá" que cuando le pedimos que avance uno, tendrá que avanzar un byte.

```
int numero;
int v[4] = { 1, 2, 3, 4 };
int *p;

{
        /* Hacemos que "p" apunte al primer elemento del vector (array) */
        p = v;      /* = que p = &v[0]; */
        numero = *( p + 3 );
}
```

Cuidado con no usar los paréntesis, ya que el operador '*' posee preferencia sobre el operador '+'.

NOTA: El tamaño de una variable tipo puntero en memoria depende del tipo de memoria que queramos direccionar (en MS-DOS 16bits (2 bytes), con memoria lineal (modo protegido) 32 bits).

Cuando usamos punteros para acceder a arrays (por ejemplo a cadenas), suele ser bastante útil declarar el puntero como variable tipo "register" ya que esto incrementa (si puede) la velocidad que ya de por sí tienen los punteros.

Ejemplo:

```
/* strlen recibe un puntero a char (cadena)
```

```
      devuelve el tamaño de la misma */
int strlen( char *s )
{
      register char *p = s;
      while ( *p ) p++;
      return ( p – s );
}
```

Los punteros y las funciones:

Hay que tener en cuenta, que C, al contrario que otros lenguajes como BASIC o PASCAL, no tiene variables pasadas a funciones por valor (copia del valor) o por referencia (dirección de la variable).

En C sólo podemos pasarle las variables por valor a una función, es decir no es la variable en sí sino un duplicado de la misma.

Para remediar ésto, hay que emplear punteros, es decir en vez de pasar una variable le pasamos otra que apunte a ella, en definitiva un puntero.
Todos los cambios que realicemos sobre la variable apuntada, como se realizan directamente en memoria, perduran una vez finalizada la función.

```
void resta2( int *numero )
{
      *numero = *numero – 2;
      /* Estupidez */
      numero = numero + 100;
}

void main( void )
{
      int n = 10;
      int *p;

      p = n;
      resta2( n );
}
```

Resultado de ejecutar el ejemplo anterior:
n => 8
p => &n
p sigue apuntando a n, porque aunque modifiquemos el valor del puntero en la función, insisto en que todas las variables se pasan por valor, es decir no hemos pasado el puntero en realidad sino una copia del valor del puntero (dirección de p en este caso).

Ejemplos:

```
/* Copia una cadena en otra */
char *strcpy( char *d, char *s )
{
      char *ptr = d;
      while ( *d++ = *s++ );
      return( ptr );
}

/* Concatena dos cadenas */
char *strcat2( char *d, char *s )
```

```
{
      char *p = d;
      strcpy( char *( d + strlen( d ) ), s );
      return( p );
}
```

Ficheros en el ANSI C:

Un fichero no es más que una serie de datos seguidos, almacenados en un soporte, que se referencia por un nombre.
(La mayoría de las funciones de alto nivel para manejo de ficheros, comienzan por 'f')

Apertura:

FILE *fopen(**<ruta>**, **<modo>**)

Abre un fichero alojado en ***<ruta>*** bajo las condiciones que le indicamos en ***<modo>***.

Si tiene éxito devuelve un puntero a FILE y en caso contrario NULL.

<ruta> Fichero a abrir
<modo> Modo de apertura:
- **r** Lectura – Abre un fichero que ya existe
- **w** Escritura – Crea un fichero (si existe lo machaca)
- **a** Añadir (append) – Si existe añade al final datos, si no lo crea.

Para permitir lectura/escritura, hay que añadir '+': r+, a+, w+

Por defecto, los ficheros se abren en modo Texto **"t"**, pero en caso de que queramos abrirlos en modo binario, añadiremos una **"b"** al modo de apertura (ejs.: rb, r+b) (En modo binario, interpreta el texto tan cual, sin embargo en modo Texto los "\n" por ejemplo los interpreta como "\r\n" luego el tamaño puede variar con el real).

Cierre:

```
int fclose( FILE *f )
```

Cierra el fichero apuntado por f, devuelve 0 si todo ha ido bien o un código de error en caso contrario. (Nota: en teoría, C cierra los ficheros abiertos al finalizar el programa, pero aún así siempre conviene asegurarse para no llevarse sustos).

Escritura de datos:

fprintf(FILE *f, ***<formato>***, ...)

Su funcionamiento es idéntico al de printf, sólo que escribirá en este caso en el fichero que le pasemos.

Ejemplo:

```
fprintf( stdout, "Hola peña! %d\n", numeraco )
```

Imprimirá por la salida estándar (normalmente la pantalla) el mensaje y la variable numeraco.

Hay que tener en cuenta, que en C, (al igual que UNIX ya que C se diseñó para crear el S.O. UNIX) todo dispositivo E/S es tratado como un fichero, luego podremos usar esta función para trabajar con cualquier dispositivo de este tipo. Nada más arrancar un programa en C bajo MS-DOS abre 5 ficheros como mínimo:

stdin	Entrada estandar (teclado)
stdout	Salida estandar (pantalla)
stderr	Salida de errores (pantalla)
stdaux	Salida auxiliar (com)
stdprn	salida impresora estandar (lpt1)

De estos 5, los 3 primeros son estándar y los 2 últimos son del MS-DOS.

Posicionamiento:

fseek(FILE *f, long **offset**, int **pos**)

Nos posiciona en el lugar que le indiquemos dentro del fichero apuntado por "f".

offset Desplazamiento relativo a partir de la posición que le indiquemos en ***pos***, puede ser positivo (hacia delante) o negativo (hacia atrás)

pos tenemos tres opciones:

SEEK_SET	A partir del principio del fichero
SEEK_CUR	A partir de la posición actual del fichero
SEEK_END	A partir del final del fichero

Ejemplo:

```
/* Quiero posicionarme en la posición penúltima del fichero "f" */
fseek( f, -1, SEEK_END );
```

long ftell(FILE *f)

Nos devuelve el desplazamiento con respecto al principio del fichero, de la posición actual (Se suele emplear para convinarlo con fseek).

Lectura de datos:

unsigned fread(void ***d**, unsigned **n**, int **tam**, FILE *f)

***d**	Puntero a la zona de memoria donde almacenaré los datos leídos.
n	Número de elementos que leeré
tam	Tamaño de cada uno de los elementos a leer.

devuelve: unsigned Número de elementos que ha conseguido leer.

Para escribir:

unsigned fwrite(void ***s**, unsigned **n**, int **tam**, FILE *f)

Funciona exactamente igual que fread, pero para escritura en vez de lectura.

Escritura/Lectura de bloques (o Registros):

Empleando las dos funciones citadas antes y una estructura, podemos leer/escribir un registro de golpe en un fichero.

Ejemplo:

```
typedef struct
{
      char nombre[20];
      int numero;
} DATOS;

FILE *fps, *fpd;
DATOS v;

/* Lectura */
fread( (void *) &v, sizeof( DATOS ), 1, fps );
/* Escritura */
fwrite( (void *) &v, sizeof( DATOS ), 1, fpd );
```

- #pragma:

#pragma es una instrucción de prepocesador que nos servirá para indicar directivas al éste. Se emplea de la siguiente manera:

#pragma **<directiva>**

<directiva> Es la directiva que le queremos indicar al preprocesador.

Por ejemplo podemos utilizar *argsused* que le indicará que es posible que no empleemos todos los argumentos que le pasamos a la función. Aunque hay muchas otras directivas (algunas específicas del compilador), así que a mirarse la ayuda.

Ejemplo:

```
#pragma argsused
void main( int argc, char **argv );
{
      (...)
}
```

Moldeado:

(**tipo**) **dato**

tipo Tipo al que se quiere convertir el dato **dato**.

Ejemplo:

```
	long n = ( long ) 30;
```

```
/*
	MOSTRAR UN FICHERO INVERTIDO
	By Jesús Arnáiz
*/

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>

/* Para evitar el "warning" */
#pragma argsused
int main ( int argc, char **argv )
{
	FILE *f;
	long total;
	char ch;

	if ( ( f = fopen( argv[1], "r" ) ) == NULL )
	{
		printf( "Error al intentar abrir el fichero\n" );
		exit(-1);
	}

	clrscr();

	/* Número de caracteres que tengo */
	fseek( f, 0L, SEEK_END );
	total = ftell( f );

	/* Voy al final del fichero */
	fseek( f, -1, SEEK_END );

	while( total )
	{
		fread( &ch, sizeof( char ), 1, f );
		printf( "-%c", ch );
		total--;
		fseek( f, -2, SEEK_CUR );
	}

	fclose( f );

	return( 0 );
}
```

Asignación dinámica de memoria:

Para hacer uso de este tipo de memoria, tenemos que declarar un variable tipo puntero al dato que queramos manejar. Se emplean, principalmente, las siguientes funciones:

```
void *malloc( size_t )
void *calloc( size_t, int tam)
void *realloc( void *ptr, size_t )
```

malloc:

Devuelve, si ha tenido éxito, un puntero (void) a la zona de memoria que se ha asignado, si no, devuelve NULL.

Como argumentos recibe:

size_t: Número de bytes que deseamos dimensionar.

Para que la llamada tenga éxito, no sólo es necesario que dispongamos de la memoria necesaria libre sino que además esta memoria debe de ser consecutiva.

calloc:
(Clear alloc)

Argumentos:

size_t: Número de elementos de tamaño 'n' que queremos que quepan en la zona de memoria.

int tam: Tamaño de estos elementos.

Produce el mismo resultado que un malloc, con la salvedad que calloc además inicializa la memoria a 0s.

Hay que tener en cuenta, que un calloc no funciona exáctamente igual que el malloc, ya que en malloc, generalmente, asignábamos un valor a size_t igual al número de elementos multiplicado por el tamaño de los mismos, nótese que con calloc esto no es necesario, ya que el tamaño de los elementos se lo pasamos en **tam**.

Ejemplo:

```
long *ptr;

/* Dos asignaciones idénticas */
/* 1 */
ptr = malloc( sizeof( long ) * 10 );
memset( ptr, 0, 10 );

/* 2 */
ptr = calloc( 10, sizeof( long ) );
```

realloc:

Reasigna una zona de memoria previamente dimensionada, o asigna una nueva en caso de que la zona que queramos redimensionar sea NULL.

Como parámetros recibe:

void *ptr: Puntero a la zona de memoria que queremos redimensionar
size_t: bytes a dimensionar (Igual que en malloc).

Retorna un puntero a la nueva zona de memoria, hay que tener en cuenta que no tendrá por qué ser la misma que antes, así que siempre tenemos que igualar el puntero al valor de esta función. También hay que pensar que realloc lo que hace realmente es liberar esa zona de memoria y meter los datos en otra (si es menor no pasa nada y si es menor, evidentemente, los trunca).

Notas sobre la asignación de memoria:

Suele ser interesante moldear el resultado devuelto por estas funciones, además así nos evitaremos warnings. También hay que pensar que si se fragmenta mucho la memoria, es posible que nos fallen las llamadas, sobre todo si trabajamos en modo MS-DOS (640K como máximo). También suele quedar más curioso si en vez de presuponer que van a funcionar las llamadas a estas funciones, evaluar el resultado con un if, si es NULL ha fallado y podemos informar al usuario y si no continuamos, ya que un intento de escritura en una zona de memoria "extraña" y se nos puede colgar el programa (o algo peor).
Otro problemilla, referente esta vez a realloc, es que si una llamada a realloc falla nos quedaremos con una zona de memoria asignada a la que nadie apunta (Error: **Null Pointer Assignement)**, para evitar esto, podemos por ejemplo realizar la llamada a realloc con una variable temporal.

Ejemplo:

```
char *ptr = NULL;
char *tmp;

ptr = ( char * ) malloc( sizeof( char ) * 10 );
tmp = ( char * ) realloc( ptr, sizeof( char ) * 20 );

/* Comprobamos que no ha fallado */
if ( tmp )
      ptr = tmp;
```

Liberar la memoria asignada:

Para liberar memoria asignada con las funciones anteriores (u otras basadas en estas) emplearemos:
free:

```
void free( void *ptr )
```

Tan sólo debemos preocuparnos de que le pasemos en ***ptr** el puntero a la zona de memoria que queremos liberar.

```c
/*
	USO DEL MALLOC
	By Jesús Arnáiz
*/

#include <stdio.h>
#include <string.h>
#include <malloc.h>
#include <conio.h>

int main(void)
{
	char * ptr = NULL;
	unsigned int n = 0;

	printf("\n Número de caracteres: ");
	scanf("%d", &n);

	if (n)
	{
		if ( (ptr = malloc( (sizeof( char ) * n) + 1 )) != NULL )
		{
			int i;

			memset(ptr, 0, (n + 1) );

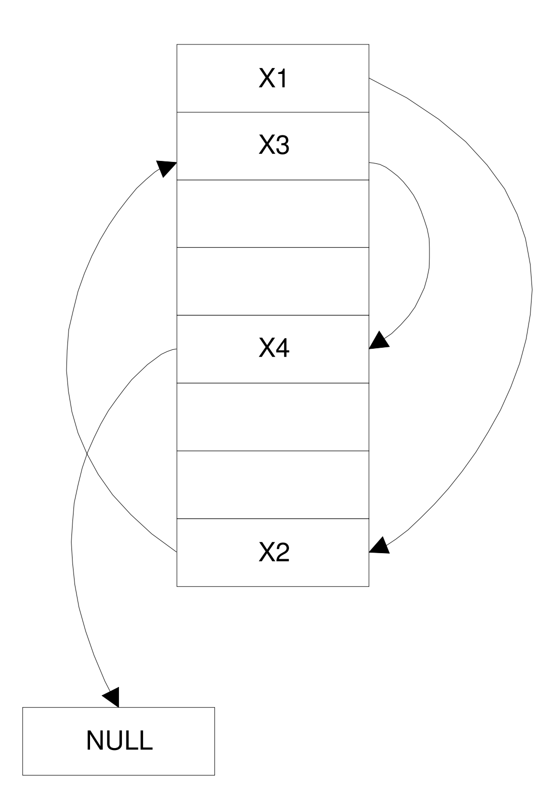
			for( i = 0 ; i < n ; i++ )
			{
				*(ptr + i) = getche();
			}
			free(ptr);
		}
		else
			printf( "Error asignando memoria\n" );
	}
	else
		printf( "Número de caracteres no válido\n");
	}

	return (0);
}
```

- **Memoria dispersa:**

Como hemos visto en el apartado anterior, el principal problema de la asignación de memoria dinámica, es que si empleamos varias llamadas, la memoria se queda fragmentada con lo que llegará un momento en el que llamadas fallen.

Para evitar esto emplearemos las técnicas de **memoria dispersa**.Lo que crearemos, será una serie de elementos en los que cada uno de ellos es capaz de direccionar al elemento que le sucede y/o al que le precede.



Ejemplo de estructura auto-referenciada.

En el ejemplo de arriba, vemos que el primer elemento (X1), apunta al siguiente (X2) y así sucesivamente hasta el último (X4) que apunta a NULL.

Listas:

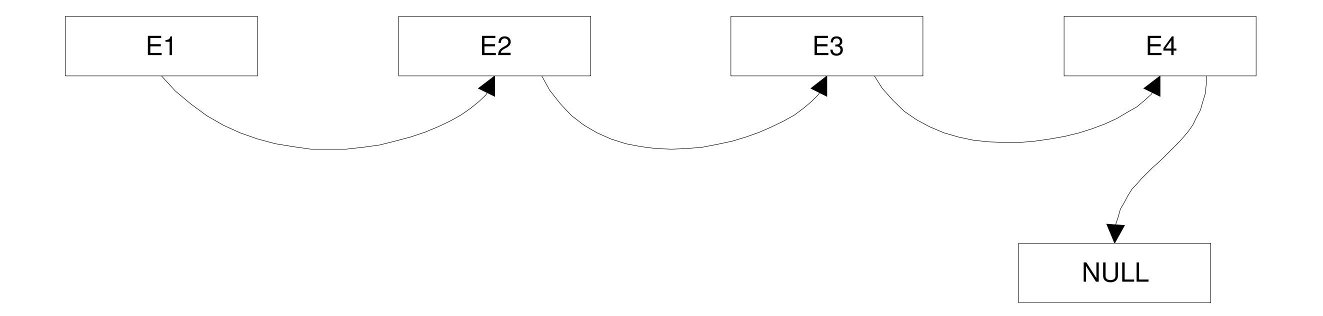
Cada elemento necesita almacenar uno o dos punteros a una estructura del mismo tipo. A este tipo de estructuras las llamamos estructuras autoreferenciadas, ya que cada una es capaz de referenciar a otra del mismo tipo.

Lista encadenada:

Cualquiera de sus nodos referencia al nodo que le siguie, pero sólo a ese. (No puedo ir en sentido contrario).

Ejemplo:

```
struct nodo
{
      int dato;
      struct nodo *siguiente;
}
```



Listas doblemente encadenadas:

Son idénticas a las anteriores, pero con la salvedad de que además de referenciar al elemento siguiente también lo hacen al anterior:

Ejemplo:

```
struct nodo
{
      int dato;
      struct nodo *siguiente;
      struct nodo *anterior;
}
```

E1 E2 E3 E4

NULL

Métodos empleados para trabajar con las listas:

Un dato que siempre tenemos que tener presente es el nodo "raiz" es decir el primer nodo de la lista.

Creación del nodo:

Emplearemos la siguiente función ("Crea_Nodo") que devolverá un puntero a nodo (el nodo que acabamos de crear) y a la que le pasaremos un dato del tipo que queremos almacenar en la lista (en los ejemplos "int").

```
PNODO Crea_Nodo( int n )
{
      PNODO nodo ;

      nodo = ( PNODO ) calloc( 1, sizeof( NODO ) ) ;
      if ( nodo )
      {
            nodo->valor = n ;
      }

      return ( nodo );
}
```

Liberar un nodo:

Para liberar un nodo, nos bastará realizar una llamada a free pasándole como parámetro el puntero al nodo que queremos liberar.

Inserción de un nodo:

Para insertar un nodo en la lista, en caso de que no esté ordenada, utilizaremos un algoritmo similar a la siguiente:

```
PNODO actual, raiz;

(...)
/* Si no hay raiz este es el primer elemento */
if ( !raiz )
      raiz = actual = nuevo ;
else
{
      actual->siguiente = nuevo ;
      actual = nuevo ;
}
```

Y para listas doblemente encadenadas:

```
if ( !raiz )
      raiz = actual = nuevo ;
else
{
      actual->siguiente = nuevo ;
      nuevo->anterior = actual;
      actual = nuevo ;
}
```

Recorrer la lista:

Para recorrer la lista, nos bastará con el siguiente bucle.

```
	while ( actual )
	{
		printf( "%d ", actual->valor ) ;
		raiz = actual ;

		/* Paso al siguiente */
		actual = actual->siguiente ;
	}
```

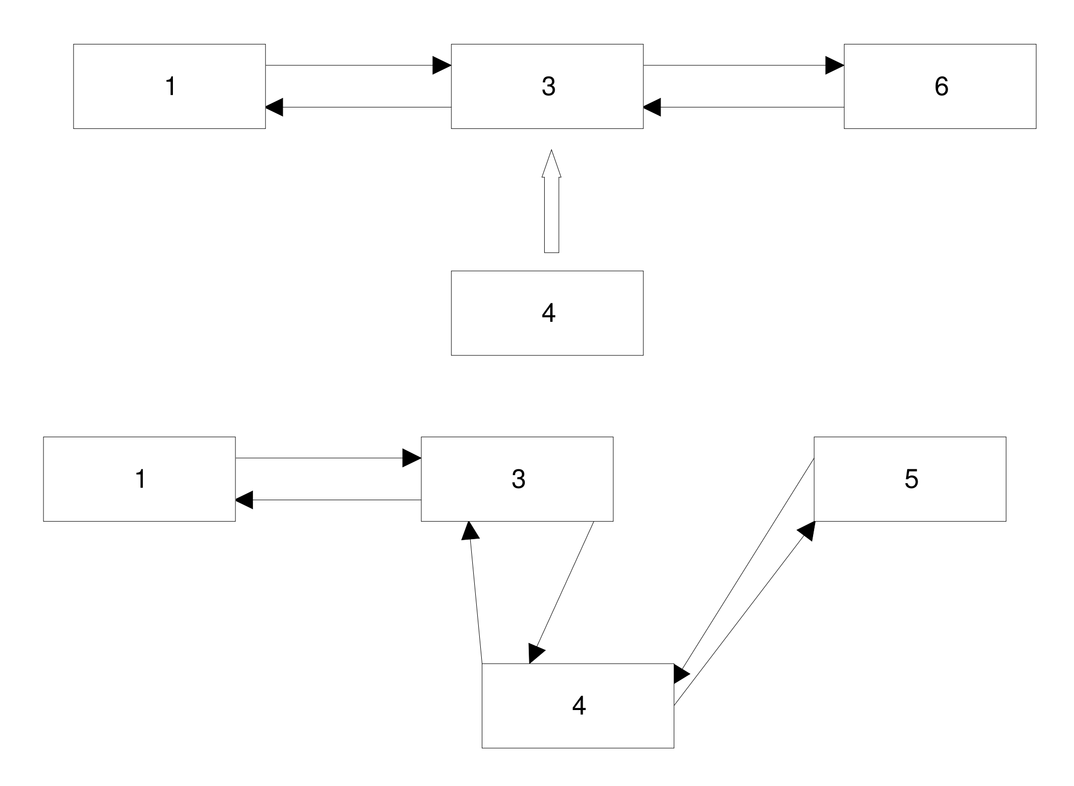
Listas ordenadas:

Para trabajar con listas ordenadas, tenemos que tener en cuenta, que a la hora de insertar caben tres posibilidades:

- Que sea más pequeño que el primero (o sea el único)
- Que sea mayor que el último
- Que esté por el medio de la lista

Si es más pequeño que el primero, nos basta con colocarlo como nodo raiz en la lista y además hacer que su siguiente sea el que hasta ahora era la raiz.

Los otros dos casos funcionan de la manera que indica el siguiente esquema.



Las funciones, básicamente, quedarán así:

/*

```c
	Localiza la posición en la que debe de ir un nodo
*/
PNODO Busca_Lugar( PNODO raiz, PNODO nuevo )
{
	PNODO tmp = raiz;
	PNODO actual = NULL;

	while ( tmp && tmp->valor <= nuevo->valor )
	{
		actual = tmp;
		tmp = tmp->siguiente;
	}

	return( actual );
}


/* Inserta un nodo en la lista */
PNODO Inserta_Nodo( PNODO raiz, PNODO nuevo, int (*cmp)() )
{
	PNODO tmp;
	if ( ( tmp = Busca_Lugar( raiz, nuevo) ) == NULL)
	{
		if ( raiz )
			raiz->anterior = nuevo;
		nuevo->siguiente = raiz;
		raiz = nuevo;

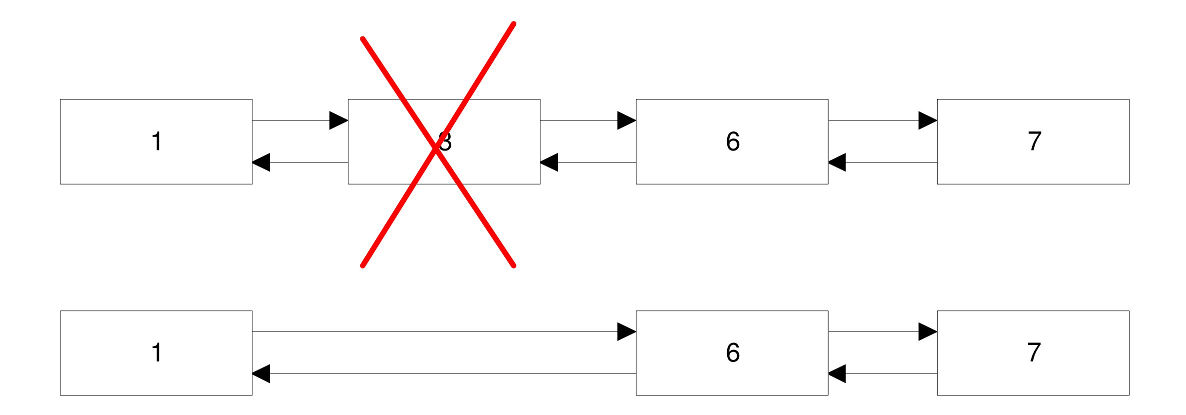
	}
	else
	{
		nuevo->siguiente = tmp->siguiente;
		nuevo->anterior = tmp;
		tmp->siguiente = nuevo;

		if ( nuevo->siguiente )
			nuevo->siguiente->anterior = nuevo;
	}

	return( raiz );
}
```

Eliminar un nodo de una lista doblemente enlazada:

Nos bastará fijarnos en el siguiente esquema, para saber como funcionará el borrado de un elemento.



Aún así, hay que tener en cuenta, que se nos presentan tres casos posibles:

- Que el nodo esté a mitad de la lista
- Que sea el primero
- Que sea el último

Para ello, nos bastará con comprobar: si el que queremos borrar tiene un nodo siguiente, al siguiente le asignamos como anterior el anterior al que queremos borrar (fijarse en el esquema para no liarse).

Y si tiene anterior, al nodo anterior "le decimos" que su siguiente es el siguiente del nodo a borrar.

Para finalizar, nos basta comprobar que si el nodo a eliminar es el raiz, como la lista no puede quedarse sin raiz, asignamos al nodo siguiente (que ahora será el primero) como raiz de la lista.

```
PNODO Elimina_Nodo( PNODO raiz, PNODO elim )
{
      if ( elim->anterior )
            elim->anterior->siguiente = elim->siguiente;

      if ( elim->siguiente )
            elim->siguiente->anterior = elim->anterior;

      if ( elim == raiz )
            raiz = raiz->siguiente;

      free( elim );

      return( raiz );
}
```

Aumentando el nivel de abstracción:

Al trabajar con las listas, deberíamos crearnos funciones lo más genéricas posible, de manera que puedan funcionar con cualquier tipo de dato, con pequeñas modificaciones.

Estructura:

```
struct nodo
{
      void *ptr_dato;
      struct nodo *siguiente;
      struct nodo *anterior;
} NODO, *PNODO;
```

Asignación de datos:

```
struct nodo tmp;

tmp->ptr_dato = ( void * ) n;
```

Comparar Datos:

```
/* Función dependiente del tipo de dato */
/* Devolveremos 0 1 o -1 según si son iguales, el primero mayor
            o menor */
/* Ejemplo con datos tipo "int" */
int Compara_Datos( void *dato1, void *dato2 )
{
      int ret = 0;

      if ( ( int ) dato1 < ( int ) dato2 )
            ret = -1
      else
            if ( ( int ) dato1 == ( int ) dato2 )
                  ret = 0;
            else
                  ret = 1;

      return ( ret );
}


/* Versión Hackers ;) */
int Compara_Datos( void *dato1, void *dato2 )
{
      int a, b;

      a = ( int ) dato1;
      b = ( int ) dato2;

      return ( ( ( a < b ) ? -1 : ( a == b ) ? 0 : 1 ) );
}
```

Como comentario, recordar que lo que se usa en el "return" de la función anterior, no es más que el "operador de condición", que funciona de la siguiente manera:

- Entre paréntesis va lo que se quiere evaluar.
- Lo que sigue a la interrogante '?' es lo que se ejecutará en caso de que se cumpla.
- En caso contrario se ejecuta lo que sigue a los dos puntos ':'

Ejemplo:

```
#define NPERSONAS 3

int n = NPERSONAS;

/* Imprime el número de personas, comprobando si hay que escribir
      "persona" (1) o "personas" (el resto) */
printf( "Hay %d persona%c\n", n,  ( n != 1 ) ? 's' : '\0' );
```

Inserción de nodos:

Para la función de Inserción de nodo emplearemos un parametro tipo puntero a función para hacer esta subrutina más independiente.

```
/* Creamos como tipo de datos RUTINA_CMP que es un puntero a función */
typedef void (*RUTINA_CMP)();


/* Idéntica a las anterioriores, pero empleando la función cmp en vez
de las comparaciones */
PNODO Inserta_Nodo( PNODO raiz, PNODO nuevo, RUTINA_CMP cmp )
{
      (...)
            while ( tmp && ( *cmp )( tmp->dato, nuevo->dato) < 1 )
            {
                  (...)
}
```

Nota: Esto de punteros a funciones es muy útil para rutinas de manejo de menús, ya que un menú podría ser un vector de estructuras del siguiente tipo:

```
struct item
{
      char titulo[TAM_MAX];
      void *( *ptr )();

}
```

Y la llamada a la ejecución de una opción así:

```
( *( menu[n].ptr ) )();
```

Colas y Pilas:

Se tratan de dos tipos de listas especiales.

Las Colas son las llamadas ***FIFO*** (First In First Out - Primero en entrar primero en salir) y su símil podría ser el de la cola del cine: el primero que llega, es el primero que sale de ella. Y para ellas utilizaremos unas funciones denominadas generalmente:

PUT -> Meter un elemento en la lista
GET -> Extraer un elemento de la lista

Las Pilas son las denominadas ***LIFO*** (Last In First Out – Último en llegar primero en salir) y funcionan exactamente igual que una pila de bandejas, en la que la última que dejas es la primera que cojes. Sus funciones son las siguientes (os sonarán del manejo de la pila en ensamblador)

PUSH -> Meter un elemento
POP -> Extraer un elemento

Normalmente, tanto las pilas como las colas tienen una estructura denominada ***Cabecera*** o ***Head*** que suele ser de la siguiente forma:

```
typedef struct
{
      PNODO primero;
      PNODO ultimo;
      PNODO actual
} HEAD, *PHEAD;
```

A los nodos primero y último, se suelen llamar ***Head*** y ***Tail*** en las colas y ***Top*** y ***Bottom*** en las pilas.

Colas:

Como hemos dicho, las colas poseen dos funciones específicas que solemos denominar Get y Put, (por sencillez, suponemos el tipo de dato "int").

Put:

Para insertar un elemento en la cola, se pueden dar dos casos:

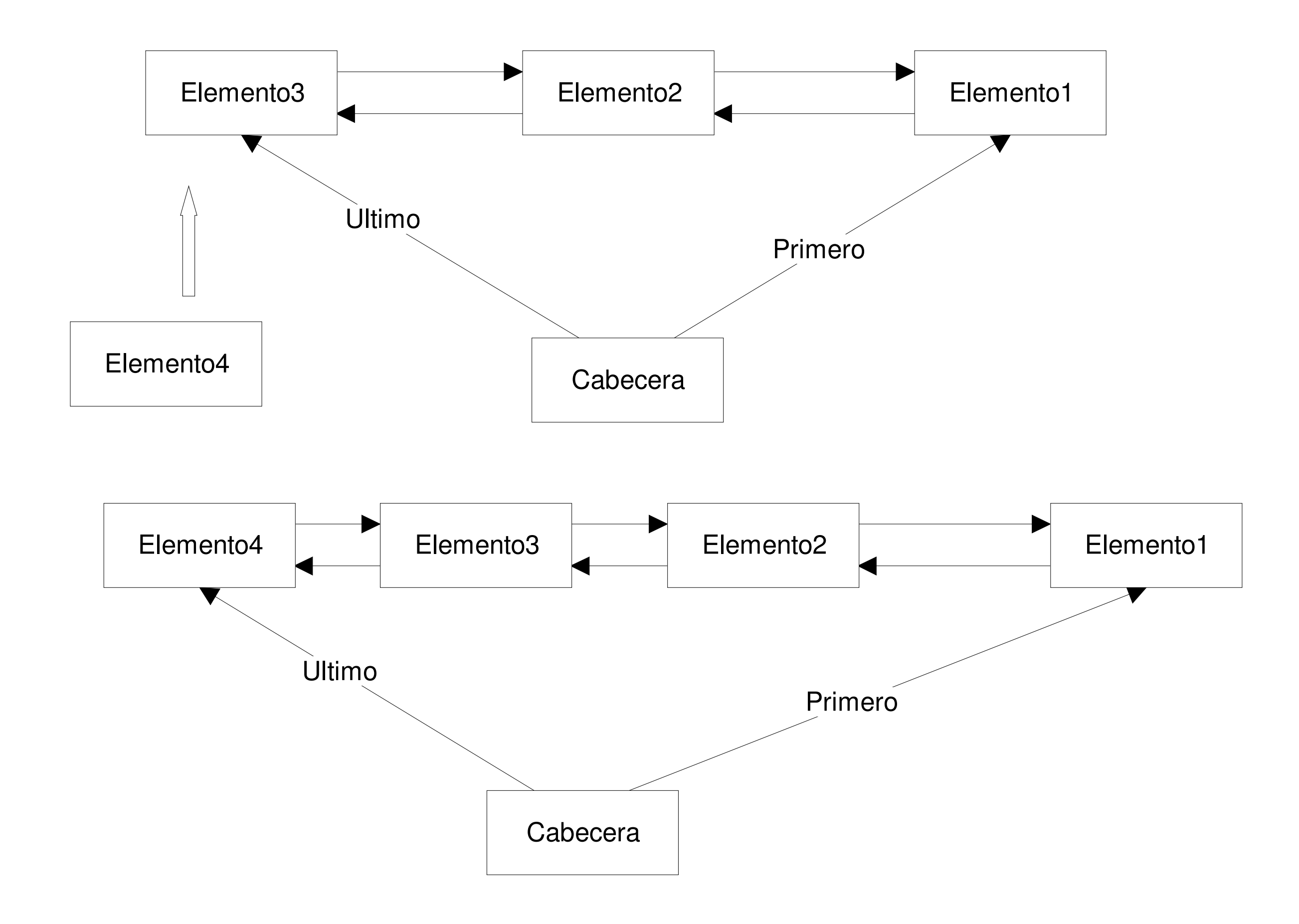
- Que no haya ningún elemento
- Que haya algún elemento

Si no hay ningún elemento, el nuevo elemento será a la vez el primero y el último de la cola, luego actualizaremos la cabecera con esos valores

Si hay algún elemento, hay que hacer varias cosas:

- El puntero "siguiente" del nodo último deberá apuntar al nuevo nodo.
- El puntero "ultimo" de la cabecera, pasará a apuntar también al nuevo nodo.

Con el esquema que expongo a continuación se puede entender el proceso a la perfección.



Y La función, en C, quedaría así:

```
PNODO Put( PHEAD Head, int n )
{
      head->actual = Crea_Nodo( n );

      if ( head->actual )
      {
            if ( !head->primero )
                  head->primero = head->ultimo = head->actual;
            else
            {
                  head->ultimo->siguiente = head->actual;
                  head->actual->anterior = head->ultimo;
                  head->ultimo = head->actual;
            }
      }

      return ( n );
}
```

Get:

Con Get extraemos el primer elemento de la cola, básicamente funciona, sacando el elemento y cambiando los valores de la cabecera.

Pasos a seguir:

- Sacar el valor del primer elemento, que será lo que devolverá la función.
- Almacenar de manera temporal un puntero al elemento a sacar.
- Hacer que el puntero "primero" de la cabecera apunte al elemento que sigue al primero.
- Liberamos el nodo.
- Hacemos que el puntero "actual" de la cabecera apunte al que ahora es el primero de la cola.
- En caso de que no haya más elementos (no haya primero), hacemos que el puntero "ultimo" de la cabecera apunte a NULL.

La función en realidad es más sencilla de lo que parece, a continuación está el código, tener en cuenta que se ha usado head->actual como puntero temporal, pero so podría haber usado una variable tipo PNODO de la misma manera.

```
int Get( PHEAD head )
{
      int n = 0;

      if ( head->primero )
      {
            n = head->primero->valor;
            head->actual = head->primero;
            head->primero = head->primero->siguiente;
            free( head->actual );
            head->actual = head->primero;
            if ( !head->primero )
                  head->ultimo = NULL;
      }

      return ( n );
}
```

Aunque lo mejor para comprobar su funcionamiento, es ver un ejemplo completo. El que pongo a continuación es básicamente el que hemos empleado con las listas, pero modificado para trabajar con Colas (los cambios son mínimos).

```
/*
      COLAS
      By J.A.
*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>

/* Estructura de los nodos */
typedef struct nodo
{
      int valor;
      struct nodo *siguiente ;
      struct nodo *anterior ;
}NODO, *PNODO ;
```

```c
/* Cabecera */
typedef struct
{
	PNODO primero;
	PNODO ultimo;
	PNODO actual;
} HEAD, *PHEAD;


PNODO Put( PHEAD head, int n );
int Get( PHEAD head );
PNODO Crea_Nodo( int valor ) ;

void main( void )
{
	HEAD head;
	char str[4];
	int n = 0;

	/* Inicializamos a 0 los valores de la cabecera -> Importante */
	memset( &head, 0, sizeof( HEAD ) );

	clrscr();

	do
	{
		/* Capturamos un número a insertar o 0 para finalizar */
		fgets( str, 4, stdin );
		n = atoi( str );
		if ( n )
			if ( !Put( &head, n ) )
				n = 0;
	} while ( n );

	/* Mostramos los elementos de la cola */
	while ( ( n = Get( &head ) ) != 0)
		printf( "%d\n", n );

}

PNODO Put( PHEAD head, int n )
{
	head->actual = Crea_Nodo( n );
	if ( head->actual )
	{
		if ( !head->primero )
			head->primero = head->ultimo = head->actual;
		else
		{
			head->ultimo->siguiente = head->actual;
			head->actual->anterior = head->ultimo;
			head->ultimo = head->actual;
		}
	}

	return ( head->actual );
}
```

```c
/* Devolvemos el valor del elemento o 0 si no hay ninguno */
int Get( PHEAD head )
{
	int n = 0;

	if ( head->primero )
	{
		n = head->primero->valor;
		head->actual = head->primero;
		head->primero = head->primero->siguiente;
		free( head->actual );
		head->actual = head->primero;
		if ( !head->primero )
			head->ultimo = NULL;
	}

	return ( n );
}


PNODO Crea_Nodo( int n )
{
	PNODO nodo ;

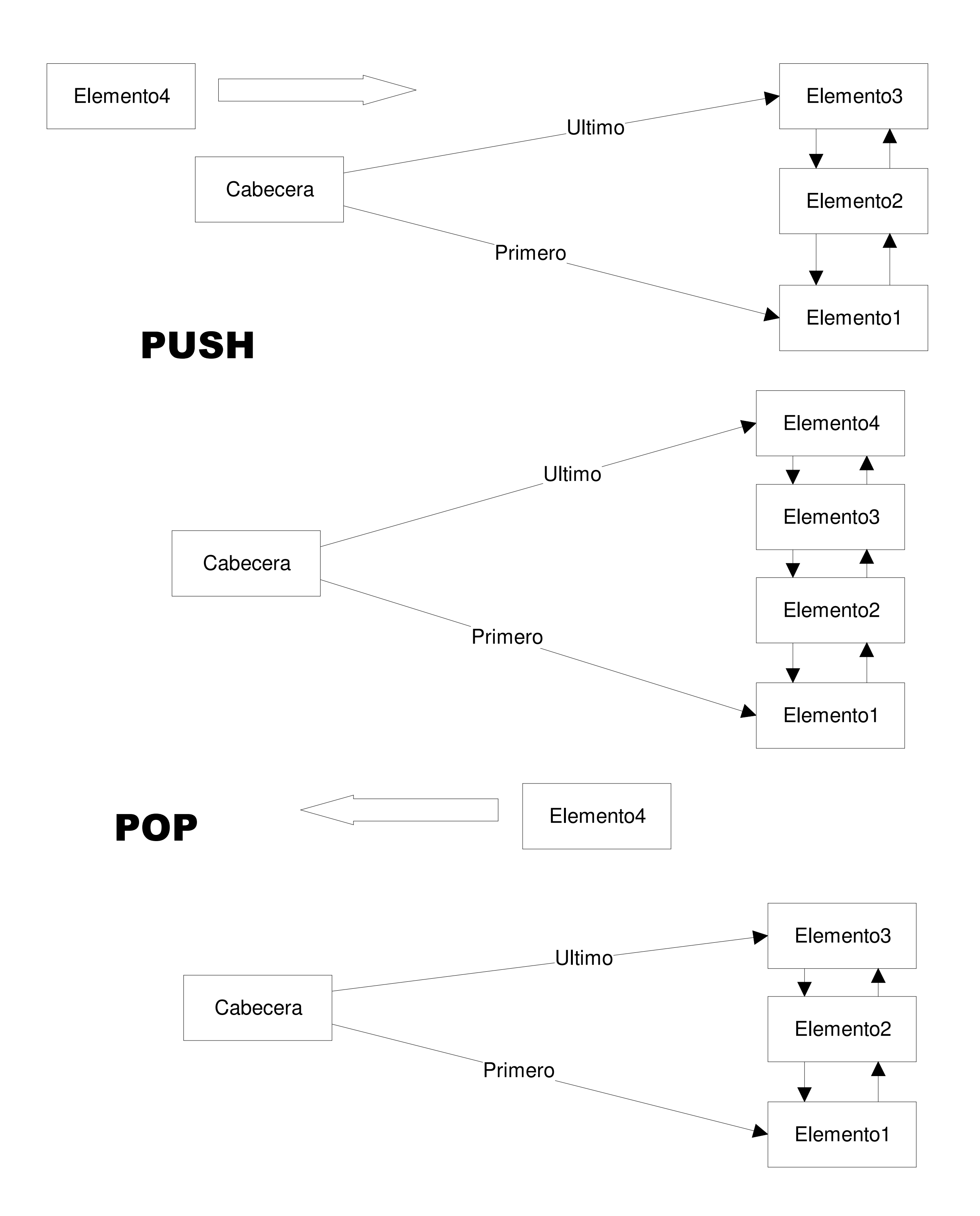
	nodo = (PNODO) calloc( 1, sizeof( NODO ) ) ;
	if ( nodo )
	{
		nodo->valor = n ;
	}

	return ( nodo );
}
```

Pilas:

Las pilas contienen dos funciones llamadas PUSH y POP. Cómo hemos dicho una pila funciona de manera análoga a una pila de bandejas o platos, luego con PUSH colocaremos un plato en la pila y con POP lo recuperaremos.

Esquemáticamente estas funciones trabajan de la siguiente manera:


Elemento4
Elemento3
Ultimo
Cabecera
Elemento2
Primero
Elemento1
PUSH
Elemento4
Ultimo
Elemento3
Cabecera
Elemento2
Primero
Elemento1
POP
Elemento4
Elemento3
Ultimo
Cabecera
Elemento2
Primero
Elemento1

Push:

Para insertar un elemento en la pila, tenemos que realizar las siguientes comprobaciones:

- Si no hay elementos, el que insertamos es el primero y el último de la pila.
- Si hay elementos, hay que hacer que tanto el puntero a "siguiente" del último de la pila como el puntero al "ultimo" de la cabecera apunten al nuevo elemento.

La función, que es muy similar a la empleada en las colas (Put), se expresa así:

```
PNODO Push( PHEAD head, int n )
{
      head->actual = Crea_Nodo( n );
      if ( head->actual )
      {
            if ( !head->primero )
                  head->primero = head->ultimo = head->actual;
            else
            {
                  head->ultimo->siguiente = head->actual;
                  head->actual->anterior = head->ultimo;
                  head->ultimo = head->actual;
            }
      }

      return ( head->actual );
}
```

Pop:

Pop nos sirve para extraer el elemento que se encuentra en la parte de arriba de la pila (es decir el último insertado). Para ello, debemos realizar las siguientes comprobaciones:

- Primero asegurarse de que haya elementos en la pila (ya que ésta puede estar vacía en cuyo caso simplemente devolveremos 0).
- Si hay elementos, extraemos el valor del último, éste será el valor que devolverá la función.
- Salvaguardaremos un puntero al último elemento
- Haremos que el puntero "ultimo" de la cabecera apunte al anterior al que era hasta el momento el último.
- Liberaremos la memoria usada por el nodo a extraer (el último).
- Haremos que el puntero "actual" de la cabecera apunte al que era hasta el momento el anterior al último.

Si nos fijamos, es muy similar a la función empleada en las colas (Get) con la salvedad de que aquí en vez de el primer elemento, extraemos el último.

```
int Pop( PHEAD head )
{
      int n = 0;

      if ( head->ultimo )
      {
            n = head->ultimo->valor;
            head->actual = head->ultimo;
            head->ultimo = head->ultimo->anterior;
            free( head->actual );
            head->actual = head->ultimo;
      }

      return ( n );
}
```

Ahora veamos el mismo programa que antes, pero aplicando pilas en vez de colas.

```
/*
      PILAS
      By ???
*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>

typedef struct nodo
{
      int valor;
      struct nodo *siguiente;
      struct nodo *anterior;
}NODO, *PNODO ;

typedef struct
{
      PNODO primero;
      PNODO ultimo;
      PNODO actual;
} HEAD, *PHEAD;

PNODO Push( PHEAD head, int n );
int Pop( PHEAD head );
PNODO Crea_Nodo( int valor ) ;
```

```c
void main( void )
{
	HEAD head;
	char str[4];
	int n = 0;

	/* Inicializamos la cabecera */
	memset( &head, 0, sizeof( HEAD ) );

	do
	{
		/* Capturamos un elemento, 0 para salir */
		fgets( str, 4, stdin );
		n = atoi( str );

		if ( n )
			if ( !Push( &head, n ) )
				n = 0;
	} while ( n );

	/* Mostramos todos los elementos de la pila */
	while ( ( n = Pop( &head ) ) != 0)
		printf( "%d\n", n );

}

PNODO Push( PHEAD head, int n )
{
	head->actual = Crea_Nodo( n );
	if ( head->actual )
	{
		if ( !head->primero )
			head->primero = head->ultimo = head->actual;
		else
		{
			head->ultimo->siguiente = head->actual;
			head->actual->anterior = head->ultimo;
			head->ultimo = head->actual;
		}
	}

	return ( head->actual );
}

int Pop( PHEAD head )
{
	int n = 0;

	if ( head->ultimo )
	{
		n = head->ultimo->valor;
		head->actual = head->ultimo;
		head->ultimo = head->ultimo->anterior;
		free( head->actual );
		head->actual = head->ultimo;
	}

	return ( n );
}
```

```
PNODO Crea_Nodo( int n )
{
      PNODO nodo ;

      nodo = (PNODO) calloc( 1, sizeof( NODO ) ) ;
      if ( nodo )
      {
            nodo->valor = n ;
      }

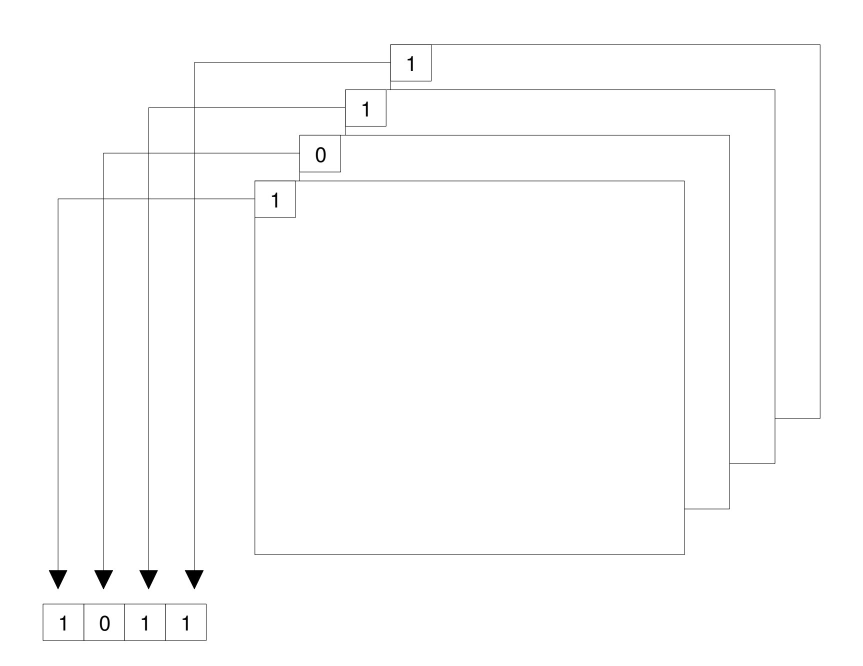
      return ( nodo );
}
```

Cabeceras de ficheros:

Para ver la importancia de las cabeceras cuando trabajamos con ficheros, veremos un caso práctico. Un visor de imágenes PCX.

Introducción al modo gráfico 13H:

Como para este ejemplo es necesario hacer uso del modo gráfico, a continuación se hace una pequeña introducción al modo 640x480 16 colores. Este modo fue uno de los empleados en las tarjetas VGA estándar, como estas tarjetas disponían de planos de 64K, y debido a que 640x480x1 (bytes) son unos 300K, es decir, no caben, se creo lo que se llama profundidad de color. Para trabajar a 16 colores necesitamos una profundidad de 4 bits. La profundidad no es más que el número de planos de 640x480 de 1 bit que tendremos para representar cada color. Si tomamos un elemento cualquiera, por ejemplo el 20, de cada plano, el conjunto de los 4 planos formará el color del pixel 20.



Formato del fichero PCX:

Cabecera + 48 bytes	Imagen	RGB (Paleta si está a 256)

Los datos de la imagen, están comprimidos con RLE (Run-Lenght Encode), que funciona de la siguiente manera:

Cada byte tiene unos bits (en este caso los dos de mayor peso) que en caso de estar a 1 significa que el siguiente dato se ha de repetir tantas veces como indiquen el resto de los bits. En caso de que no estén a 1 se trata de un pixel sin más.



La cabecera del fichero está formada por 128 bits, y la leeremos con la siguiente estructura.

```
	typedef struct
	{
		BYTE Header; /* Es un PCX? */
		BYTE Version; /* Tipo de PCX (Color16 ...) */
		BYTE Encode; /* Est  comprimido o no */
		BYTE BitPerPix; /* Número de bits para representar un pixel */
		unsigned X1, Y1, X2, Y2; /* Posición */
		unsigned Hres, Vres; /* Res Hor y Ver. àptima para visualizrla */
		char Paleta[48];
		BYTE Vmode; /* No se usa, era el modo de v¡deo de la imagen */
		BYTE NumofPlanes; /* Número de planos de bits para representar un punto */
		unsigned BytesPerLine; /* N£mero de bytes para analizar cada l¡nea */
		BYTE relleno[60]; /* No usado */
	} PCXFileHeader;
```

Veamos ahora el programa completo:

```
/*
	Visor de PCX

	Coder: J.A.
*/

#include <stdio.h>
#include <string.h>
#include <dos.h>
#include <conio.h>
#include <malloc.h>
#include <graphics.h>
```

```
#define PCXHdrTag 10
#define STRIPSIZE 50
#define PCXMono 0
#define PCXColor16 1
#define PCXColor256 2
#define MAX_BUF 32000

typedef unsigned char BYTE;

typedef struct
{
      BYTE Header; /* Es un PCX? */
      BYTE Version; /* Tipo de PCX (Color16 ...) */
      BYTE Encode; /* Est  comprimido o no */
      BYTE BitPerPix; /* Número de bits para representar un pixel */
      unsigned X1, Y1, X2, Y2; /* Posición */
      unsigned Hres, Vres; /* Res Hor y Ver. àptima para visualizrla */
      char Paleta[48];
      BYTE Vmode; /* No se usa, era el modo de v¡deo de la imagen */
      BYTE NumofPlanes; /* Número de planos de bits para representar un punto */
      unsigned BytesPerLine; /* N£mero de bytes para analizar cada l¡nea */
      BYTE relleno[60]; /* No usado */
} PCXFileHeader;

char *ScreenLines[480];
char huge *ScanLine[480];

FILE *ReadPCXHeader( PCXFileHeader *, char * );
FILE *ReadPCXFile( FILE *, int, int );
int LeeLineaPCX( int, int, char huge *, int );
void PonLineaPixels( int, int, char huge *, int );
void PonModoGrafico( void );
void QuitaModoGrafico( void );
void InitScreenLines( void );
void VgaPlane( int );

int main( int argc, char **argv )
{
      FILE *fp;
      PCXFileHeader hPCX;
      unsigned int ancho, alto, bytes;
      register int i;
      int ret;

      if ( argc < 2 )
            return 1;
      fp = ReeadPCXHeader( &hPCX, argv[1] );
```

```
	if ( fp )
	{
		ancho = hPCX.X2 - hPCX.X1 + 1;
		ancho = ( ancho + 7 ) & ~7;
		alto = hPCX.X2 - hPCX.X1 + 1;
		bytes = hPCX.BytesPerLine * hPCX.NumOfPlanes;
		PonModoGrafico();
		InitScreenLines();
		ReadPCXFile( fp, alto, bytes );

		for ( i = 0; i < alto; i++ )
		{
			PonLineaPixels( i, ancho / 8, scanline[i], hPCX, NumOfPlanes );
			farfree( scanline[i] );
		}

		getch();
		QuitaModoGrafico();
	}
	return( ret );
}

FILE *ReadPCXHeader( PCXFileHeader *, char * )
{
	FILE *fp;

	if ( ( fp = fopen( filename, "rb" ) ) == NULL )
		return ( NULL );

	if ( ( fread( hPX, 1, sizeof( PCXFileHeader ), fp )
		!= sizeof( PCXFileHeader ) )
	{
		fclose( fp );
		return( NULL );
	}

	if ( ( hPCX = Header ) != PCXHeaderTag )
	{
		fclose( fp );
		return( NULL );
	}
	return( fp );
}

void PonLineaPixels( int fila, int bytes, char huge *ptr, int nplanos )
{
	register i;
	int offset, inc;
	int planos[4] = { 1, 2, 4,  8 };

	for ( i=0 ; i<nplanos ; i++ )
	{
		offset = i * bytes;
		inc = (( bytes % 2 ) == 0 ) ? 0 : i;
		VgaPlane ( planos[i] );
		memcpy( screenlines[fila], ptr + offset + inc , bytes );
	}
}
```

```
int LeeLineaPCX( char huge *ptr, FILE *fp, unsigned int bytes )
{
      int n = 0;
      register short c;
      unsigned rep;
      do
      {
            c = getc( fp );
            if ( c == EOF )
                  return  0;

            if ( ( c & 0xC0) == 0xC0 )
            {
                  rep = x & ~0xC0;
                  c = getc(  fp );

                  if ( c == EOF )
                        return 0;

                  while  ( rep-- )
                        ptr[ n++ ] = c;
            }
            else
                  ptr[ n++ ] = c;
      }
      while ( n < bytes );

      return n;
}

FILE *ReadPCXFile( FILE *fp, int lineas, int bytes )
{
      register int i;

      for( i = 0 ; i < lineas ; i++ )
      {
            ScanLine[i] = ( char huge * ) farmalloc ( 80 * 4 );

      }
}

void PonLineaPixels( int fila, int bytes, char huge *ptr, int bytes )
{
      register int i;
      int offset, inc;
      int planos[4] = { 1, 2, 4, 8 };

      for ( i = 0 ; i < nplanos ; i++ )
      {
            offset = i * bytes;
            inc = ( ( bytes % 2 ) == 0 ) ?0:i;
            VgaPlane( Planos[i] );
            memcpy( ScreenLines[fila], ptr + offset + inc, bytes );
      }
}
```

```
/*
      Hace uso de un driver creado con bgiobj
      MS-DOS:
            bgiobj egavga.bgi
*/
void PonModoGrafico( void )
{
      int gdriver = DETECT, gmode, errorGr;

      errorGr = registerbidriver( EGAVGA_driver );
      if ( errorGr > -1 )
            initgraph( &gdriver, &gmode, "" );
      else
            printf( "Error al intentar inicializar el modo gr fico\n" );
}

void QuitaModoGrafico( void )
{
      orecrtmode();
}

/*
      Inicializa el puntero al comienzo de cada l¡nea de la
      memoria de v¡deo

      hay 80 bytes por l¡nea (640 / 8 = 80)
*/
void InitScreenLines( void )
{
      int i;

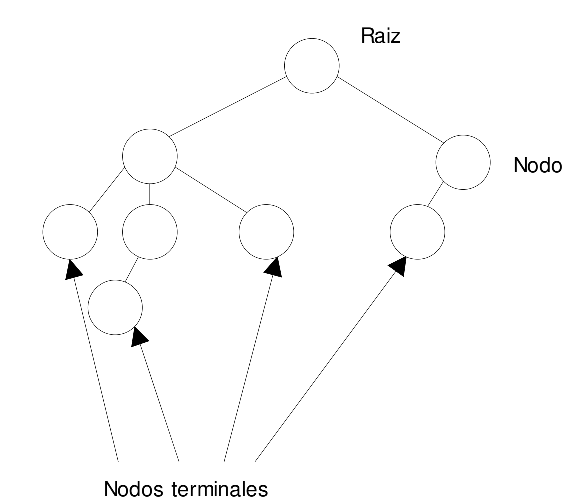
      for ( i = 0 ; i < 480 ; i++ )
            ScreenLines[i] = MK_FP( 0xA000, i * 80 );
}

/*
      Conmuta el plano
      REG 0x3C4 -> 0x02 -> Para cambiar de plano
      REG 0x3C5 -> N£mero de plano al que quiero que cambie
*/
void VgaPlane( int n )
{
      outportb( 0x3C4, 0x02 );
      outportb( 0x3C5, n );
}
```

Árboles:

Un árbol esuna estructura recursiva de datos en la que cada nodo puede tener 0, 1 o más descendiantes y así sucesivamente.

Los árboles, están ordenados de alguna manera (Según se van insertando).



Los nodos que están por debajo de otro son sus **descendientes.**

Los que están justo debajo son sus **descendientes directos.**

Antecesores: los nodos que tiene por encima.

Antecesores directos: los nodos que están un nivel por encima.

Árbol binario: **aquel en el que todos sus nodos tienen como mucho dos hijos. (descendientes directos).**

Árboles multicamino: alguno de sus nodos tiene más de dos nodos hijos.

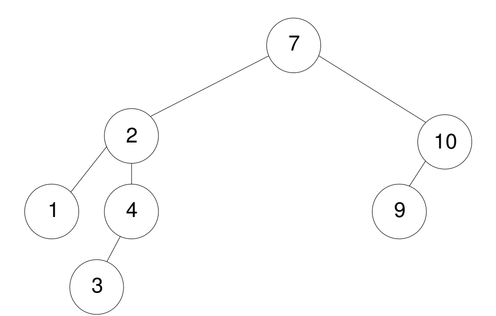
Grado del árbol: **nº máximo de hijos que tiene un nodo.**

Altura: Número de niveles.

Longitud de camino de un nodo: **nº de saltos que ha de dar desde el nodo raíz para posicionarse en ese nodo.**

Árboles binarios:

Bajando por la derecha de un nodo, son mayores a éste, y por la izquierda menores.



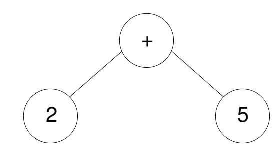
Un árbol está **equilibrado**, cuando el subárbol izquierdo y el derecho difieren como mucho en un nodo.

Árbol degenarado: no cumple la **regla de las alturas:** en el último nivel no debería haber por encima de él más alturas que las imprescindibles.

Árbol balanceado: (AUL) cada vez que se inserta un elemento se registra para no violar la regla de las alturas.

Árbol equilibrado: cuando el subárbol izquierdo y el derecho difieren como mucho en un nodo.

Lectura de un árbol:



Pre-Orden: (Prefija) raíz – izquierda – derecha "+23"

In-Orden: (Infija) izquierda – derecha – raíz "2+3"

Post-Orden: (Postfija) izquierda – derecha – raíz "23+"

Árbol B+:

Se emplean cuando no sabemos el número de hojas (nodos) que tendrá. En vez de crecer de la raíz hacia abajo, en este caso crece de las hojas hacia la raíz.

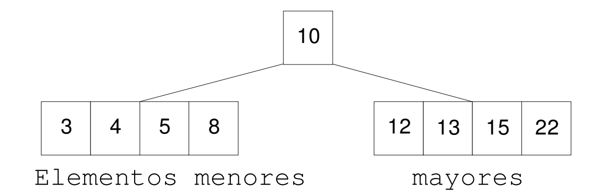
Un uso típico es el de los índices de las bases de datos.

Ejemplo:

Defino la página de por ejemplo de 9 elementos.

3	4	5	8	10	12			

Cuando se llena la página, se procede de la siguiente manera: el elemento del centro lo colocamos como padre de dos hijos: los de la izquierda (menores) y los de la derecha(mayores).



Ejemplo práctico de árbol.

```
/*
	21/12/99

	Jesús
*/

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>

typedef struct nodo
{
	int info;
	struct nodo *left;
	struct nodo *right;
} NODO, *PNODO;

PNODO MakeTree( int n );
void SetLeft( PNODO p, int n );
void SetRight( PNODO p, int n );

/* Formas de recorrer el árbol */
void PreOrden( PNODO raiz );
void InOrden( PNODO raiz );




void main( void )
```

```
{
      PNODO tree = NULL;
      PNODO p, q;
      int num;
      char str[21];

      do
      {
            fgets( str, 21, stdin );

            if ( ( num = atoi( str ) ) != 0 )
            {
                  if ( !tree )
                        tree = MakeTree( num );
                  else
                  {
                        p = q = tree;

                        while (  ( num != p->info ) && ( q != NULL ) )
                        {
                              p = q;

                              if ( num < p->info )
                                    q = p->left;
                              else
                                    q = p->right;
                        }

                        if ( num == p->info )
                              printf( "Error, %d ya existe\n", num );
                        else
                        {
                              if ( num < p->info )
                                    SetLeft( p, num );
                              else
                                    SetRight( p, num );
                        }
                  }
            }
      } while ( num );

      PreOrden( tree );
}

/* Izquierda Derecha Raiz */
void PostOrden( PNODO raiz )
{
      if ( raiz )
      {
            PostOrden( raiz->left );
            PostOrden( raiz->right );
            printf( "%d\n", raiz->info );
      }
}




/* Raíz Izquierda Derecha */
```

```
void PreOrden( PNODO raiz )
{
      if ( raiz )
      {
            printf( "%d\n", raiz->info );
            PreOrden( raiz->left );
            PreOrden( raiz->right );
      }
}

/* Izquierda Raíz Derecho */
void InOrden( PNODO raiz )
{
      if ( raiz )
      {
            InOrden( raiz->left );
            printf( "%d\n", raiz->info );
            InOrden( raiz->right );
      }
}


PNODO MakeTree( int n )
{
      PNODO p;

      p = ( PNODO ) calloc( 1, sizeof( NODO ) );

      if ( p )
            p->info = n;

      return ( p );
}

void SetLeft( PNODO p, int n )
{
      if ( p == NULL )
            printf( "Inserci¢n NO efectuada\n" );
      else
      {
            if ( p->left != NULL )
                  printf( "Inserci¢n NO efectuada\n" );
            else
                  p->left = MakeTree( n );
      }
}
```

```
void SetRight( PNODO p, int n )
```

```
{
      if ( p == NULL )
            printf( "Inserci¢n NO efectuada\n" );
      else
      {
            if ( p->right != NULL )
                  printf( "Inserción NO efectuada\n" );
            else
                  p->right = MakeTree( n );

      }
}
```

En el programa, podemos ver que las funciones dedicadas a recorrer el árbol, trabajan de forma recursiva, es decir, son funciones que se llaman a sí mismas.

Algunas notas sobre la práctica:

Aquí voy a intentar reunir algunas de las ideas que se han expuesto en clase sobre la práctica (Programa para manejar bases de datos tipo DBASE).

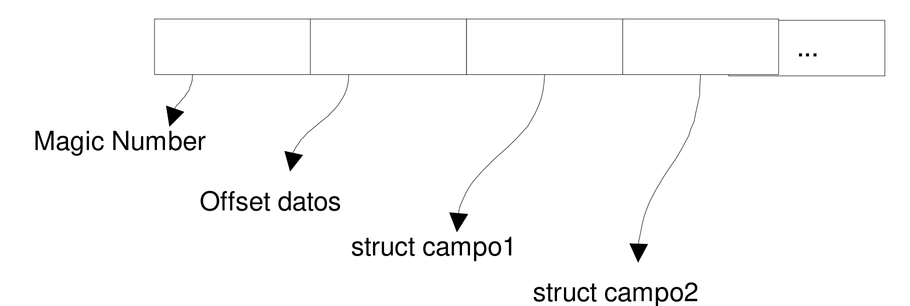
Cabecera del fichero:

Cada tabla (o base de datos) estará contenida en un fichero, este fichero debe tener como estructura básica la siguiente:

Cabecera	Datos

Siendo los datos, los datos en sí del fichero y la cabecera una serie de parámetros que definiremos previamente y que podremos leer/escribir empleando una estructura (struct).

Pasemos a ver con más detalle que campos o parámetros podría contener la cabecera (estos son sólo un ejemplo, se pueden poner muchos más).



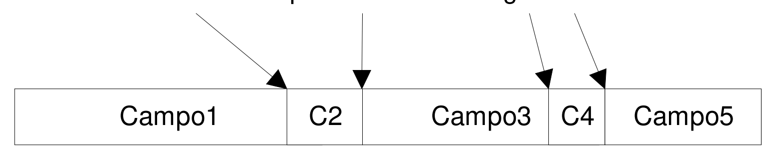
El Magic Number es, o número mágico, no es más que un número que emplearemos para certificar que ese fichero es del tipo que nosotros queremos manejar. (Dándole el valor que más nos guste).

El Offset datos es un valor que nos indica a partir de que posición del fichero empiezan los datos en sí del mismo.

Los “struct campo” serán unas estructuras que hemos creado previamente, para contener los datos que identifican a un campo dentro del registro, por ejemplo algunos podrían ser:

```
Tipo: (Entero, Real, Fecha ...)
Longitud: 10, 1, 30 ...
Decimales: 0, 2 ...
Nombre: "Dirección", "Nombre" ...
Offset: 10, ... (Desplazamiento con respecto al registro)
```

El offset dentro del registro, funcionaría así:



Es decir si leemos un registro en un buffer (un array de chars) cuantos char a la derecha del principio, se encuentra cada campo.

La cabecera debería contener también valores como el número de campos que contiene la tabla (para saber cuantos leer luego).

Notas sobre el fichero en sí:

Al añadir registros, deben de ir al final del mismo. Si lo que hemos hecho ha sido modificarlo, lo dejamos donde estaba.

El fichero siempre contiene algo (como mínimo la cabecera).

Para manejar el fichero, y si quieres complicarte la vida, puedes emplear técnicas de caché, ficheros de índice y un sinfín de cosas.

Sobre la pila de ventanas:

Si vamos a implementar rutinas de pantalla con pila de ventanas, hay que tener en cuenta unas cuantas cosas. Para empezar, las funciones de esta pila varían levemente, PUSH se quedaría como estaba, pero la función POP debería recibir un parámetro más tipo entero en el cual especificar un identificador de ventana (o 0 para sacar la que está encima).

Los ids. de las ventanas comenzarán en 1 (para poder referirnos a la de arriba con 0) y para saber si un id. está elegido o no podemos emplear un bitset:

```
char bitset[4]; /* Para tener 32 identificadores distintos */
```

(Los id. sin seleccionar están a 0 y los seleccionados a 1).

Para manejar esto, debemos crearnos macros que a partir de un carácter y un bit nos devuelvan su valor boleano (1 o 0).

Captura de una tecla:

Veamos este sencillo algoritmo: se trata de capturar una pulsación del teclado, pero en MS-DOS los caracteres normales ocupan un char y los especiales 2 devolviendo a la primera captura de getch() 0. Para ello, crearemos una función que devuelva un entero, los caracteres normales serán de la forma:

0x0091 -> ('a' o ...)

y los especiales

0xA000 -> ("cursor arriba" ...)

veamos como implementarlo en lenguaje C:

```
int getkey( void )
{
      int tecla = 0;

      tecla = getch();
      if ( !tecla )
            tecla = getch() << 8; /* Los 8 bits del primer byte */

      return ( tecla );
}
```

luego, dentro del programa deberíamos crearnos "defines" con las teclas a emplear como el "Intro" "Escape" y etc.

Estructura de un campo:

Esta podría ser la estructura de un campo si contemplamos como tipos genéricos la cadena el entero y el real.

```
struct
{
      int tipo;
      int len;

      union
      {
            char *str;
            int n;
            double v;
      } dato;
}
```

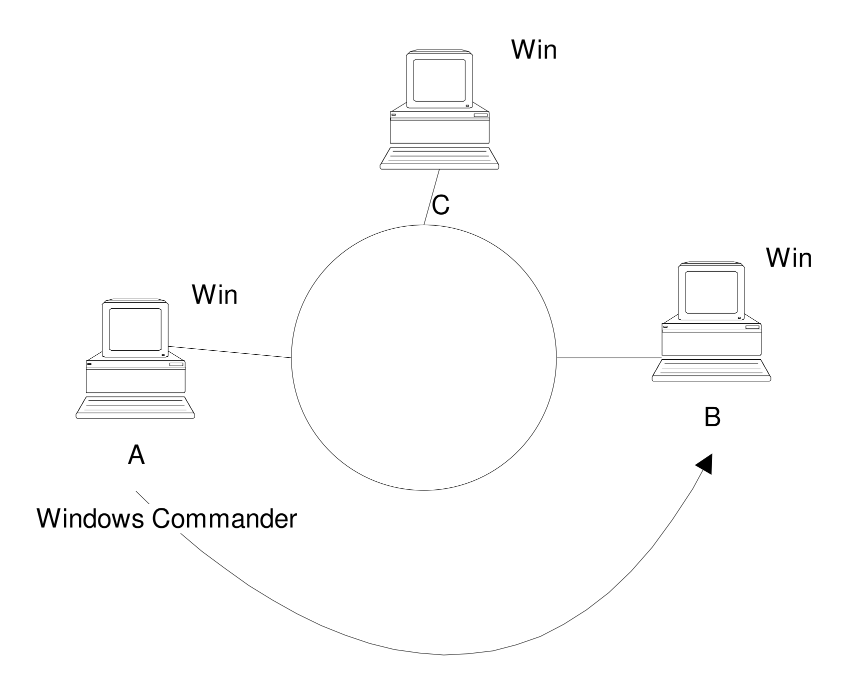
Pero está claro que el mejor método (y en realidad el más sencillo de manejar), es el de almacenar todos los datos de un campo como puntero a char (cadena), ya que así podemos escribir y leer de los ficheros los datos, y para trabajar con ellos no tenemos más que usar las funciones que nos brinda el C para conversión (librería "ctype.h").

Campos tipo fecha:

Ya hemos llegado a la conclusión que debemos almacenar todos los campos como cadenas (punteros a char), pero el de fecha, nos es mucho más práctico almacenar al contrario de cómo lo escribimos aquí en España (en vez 18102000 – poner 20001018) es decir, de la forma AAAAMMDD ya que esto nos automatiza las comparaciones entre fechas.

Recursos Distribuidos:

Tenemos, por ejemplo una red homogénea de sistemas tipo Windows:



Desde el ordenador B queremos ejecutar el "Windows Commander" que se encuentra en la máquina A, para ello una forma cutre pero que de momento nos serviría, sería la que muestra el esquema, que es simplemente trasladar el Windows Commander a la memoria principal de B para ejecutarlo en local desde B.

Pero los problemas empiezan cuando esta red no es homogénea, sino heterogénea (como son la mayoría de las redes, por ejemplo Internet). Bueno, para ello deberíamos crear una interfaz mediante la cual yo diese una orden de ejecución desde B con una serie de argumentos a un programa que se haya en A y A debe devolver el resultado obtenido, sin tener que preocuparse B del cómo.

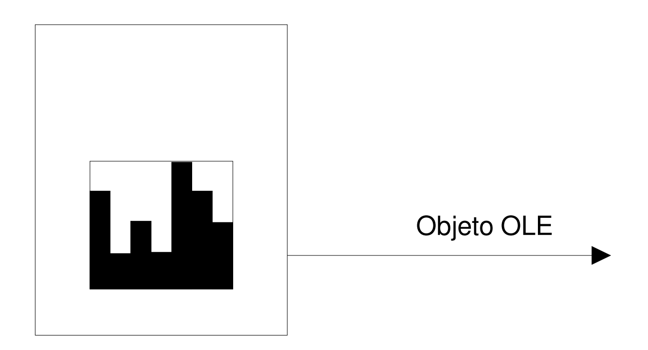
Para resolver este tipo de problemas han aparecido un par de "estándares":

OLE/COM y CORBA.

OLE/COM:

Fue el primero de los dos en aparecer, nació de la mano de Microsoft debido a las necesidades. OLE (Object Linked and Embeded).

Un ejemplo de uso de OLE puede ser un documento de texto que contenga una gráfica (de Excel p.ej.), el procesador de textos no sabe como se representa, pero el objeto en si mismo sí. Ya que al insertar el objeto se guardan los datos y los procedimientos necesarios para manejarlo.



El OLE/COM acabó generando DCOM (Distributed Component Object Module) ya que OLE sólo podía trabajar con servicios de la propia máquina.

DCOM:

Puede trabajar lo mismo que OLE, pero al contrario que este, puede hacerlo en máquinas remotas. A raíz de este último se creo el "engendro" del ActiveX (Que por cierto además de no trabajar más que con máquinas Windows, no acaba de funcionar bien).

ActiveX:

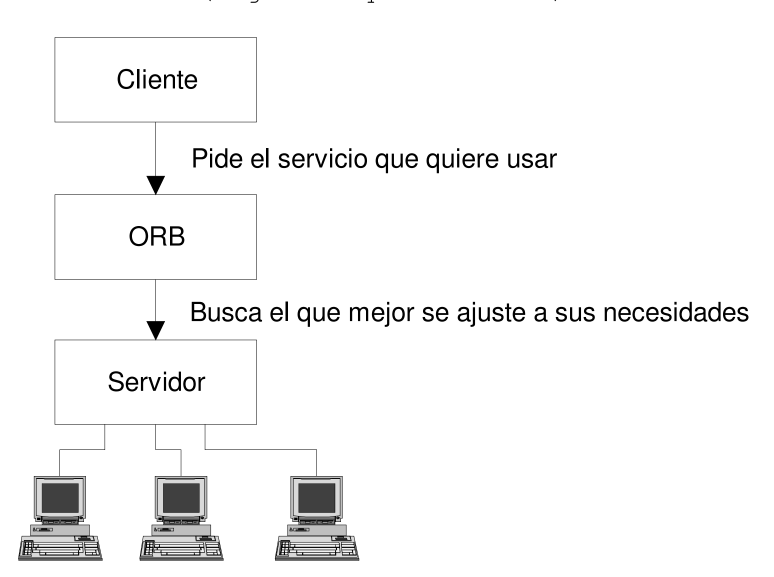
ActiveX al ser propiedad de Microsoft no está muy extendido, debido a las licencias necesarias para su uso.

CORBA:

CORBA (Common Object Request Broker Access) no es propietario (no es de nadie) y está implementado en prácticamente todos los SO conocidos: AS/400, Solaris, Linux, Windows ...

Diferencias esenciales entre CORBA y ActiveX:

- Un objeto OLE es su propio servidor.
- CORBA sin embargo pone una capa intermedia entre el Cliente y el Servidor, llamada ORB (Object Request Broker)



- Para manejar el ActiveX tienes un interfaz fijo: llamadas a Iunknow que devuelve una VTABLE (Tabla Virtual de procedimientos que posee el servidor de ActiveX).
- El ActiveX es sencillo de implementar en SoftWare ya diseñado, eso sí con POO (Programación Orientada a Objetos), ya que si no es POO es muy complicado.
- CORBA emplea unos mecanismos llamados IDL (Interface Definition Language) para implementar tanto un cliente como un servidor. De esta manera, al crear el cliente se define una clase pública que se exportará y se incluirá en el cliente para poder acceder a los métodos de la misma. Esta manera de trabajar, entre otras cosas, hace que si creamos una nueva versión del Servidor, Clientes viejos puedan seguir trabajando con él.
- ActiveX puede trabajar con C++, VB, quizá C y puede que algún lenguaje más.
- CORBA admite casi cualquier lenguaje de programación: FORTRAN, COBOL, C, C++ ... (El IDL es un lenguaje de script similar a C++ o Java).
- ActiveX no permite herencia es decir si tenemos un ActiveX que haga algo y queremos implementar otro que haga ese algo y un poco más, tenemos que reescribirlo entero.
- CORBA si permite herencia con las ventajas que esto supone. (Sólo reescribiríamos lo nuevo, lo demás podríamos hacer que lo heredase).

Interfaces gráficas de usuario

Los primeros interfaces eran modo texto, y estaban preparados para la programación lineal (secuencial). Esto cambió con la aparición de un nuevo periférico, el ratón

El primer interfaz de usuario lo creó Xerox en Palo Alto, tiempo después, Machitosh se basó en esto implementando un interfaz sólo gráfico.

Vamos a analizar dos tipos distintos de interfaces gráficos: **Windows** (3.1, 95, NT, 98 ...) para el cual necesito una copia de Windows en cada máquina para poder trabajar, y cuya "multitarea" es un híbrido entre la de tipo cooperativa y la preventiva.

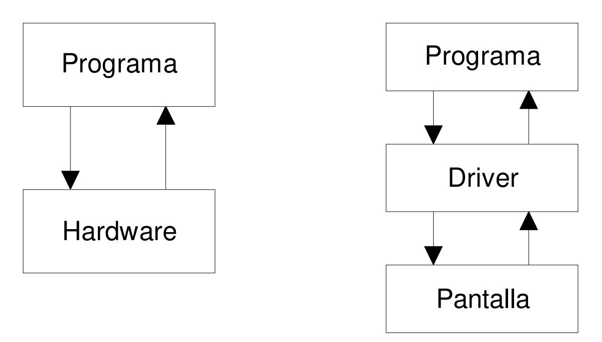
X-Window:

Trabaja con la familia de los UNIX (Ultrix, Solaris, Linux ...), se desarrolló en el MIT (Massachuses Institute of Technology).

Está basada en el modelo cliente/servidor, de manera que el servidor ejecuta el programa y devuelve el resultado al cliente. Se pretende que los programas funcionen con casi cualquier hardware.

Para esto, es necesario un diseño por capas, ya que en un principio, los interfaces gráficos, tenían que trabajar directamente con el hardware de vídeo de la máquina.

La forma antigua y la moderna de trabajar están esquematizadas en la siguiente imagen.



De esta manera, el programa llama a unas funciones genéricas que el driver se encargará de traducir para que la pantalla las entienda. En teoría sería el fabricante el que sacase drivers para sus tarjetas.

Con respecto a los interfaces gráficos, IBM creó unas normas:

SAA (System Aplication Architecture) y CUA (Common Used Access):

Las cuales, son seguidas por casi todos los interfaces gráficos de usuario. En ellas se especifica:

- Como han de ser las ventanas
- Qué contienen
- En que orden aparecen las cosas

Con esto se obtiene como ventaja identificar rápido los items, pero vemos varias desventajas: se consumen muchos más recursos, muchas veces muestran demasiada información, supone una forma más complicada de trabajar para los programadores.

Rutinas de búsqueda:

Se emplean para localizar un dato en un conjunto de ellos, que pueden estar o no ordenados.

Tipos generales de búsquedas:

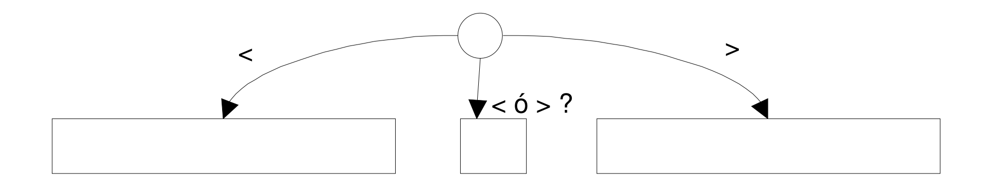
Son tres: Secuencial, Binaria y Secuencial-Indexada. Su manera de trabajar es la siguiente:

Secuencial:

Teniendo n elementos, efectúo la búsqueda posicionándome en el primero y recorriéndolo hasta el final o hasta que encuentre el dato. Para este tipo de búsqueda los datos pueden o no estar ordenados.

Binaria:

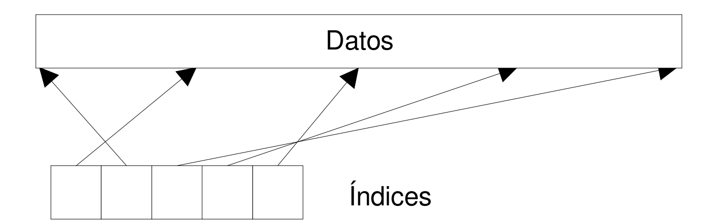
Se trabaja con un conjunto ordenado de datos, y lo que se hace es subdividir el conjunto de ellos en dos partes, comparando y pasando a buscar en una de las dos (según si es mayor o menor).



Secuencial Indexada:

Es una mixta entre las dos anteriores. Funcionan sobre datos ordenados o desordenados.

Lo que se hace, es tener por un lado los datos, y por otro lado n (número de registros) claves ordenadas que referencian a esos datos. De este modo, para buscar, primero miro en el índice (que está ordenado), y luego paso a leer (con la referencia) los datos.



Implementación de los algoritmos:

Búsqueda Secuencial:

```
/* Búsqueda secuencial */

#include <stdio.h>

int m[] = { 6, 4, 8, 4, 6, 7 };

int busca( int m[], int n, int c );

int main( void )
{
	char str[11];
	int n;

	fgets( str, 11, stdin );          Número de elementos
	n = atoi( str );                ________________________
	printf( “%d\n”, busca( m, sizeof( m ) / sizeof( m[0] ), n );

	return ( 0 );
}
```

```
int busca( int m[], int n, int c )
{
	int r = -1;
	int i;

	/* Buscamos hasta o bien encontrar el elemento o llegar al final */
	for ( i = 0 ; ( i < n ) && ( r == -1 ) ; i++ )
		if ( m[i] == c )
			r = i;

	return( r );
}
```

Notas:

El número de elementos que posee el vector, se ha obtenido dividiendo el tamaño total del vector (sizeof(m)) entre el tamaño de uno de sus elementos (sizeof(m[o])), por poner un ejemplo sencillo: si el vector ocupa 8 y cada uno de sus elementos 2, tiene 4.

El algoritmo de búsqueda (al igual que en el siguiente ejemplo) devuelve (-1) si no lo encuentra o la posición del vector (comenzando en 1 no en 0) en dónde está la primera ocurrencia.

Búsqueda Binaria:

```
int binaria( int n, int m[], int tam )
{
	int iz = 0, de = tam - 1;
	int medio;

	while( iz <= de )
	{
		medio = ( ( iz + de ) / 2 );

		if ( m[medio] < n )
			iz = medio + 1;
		else
		{
			if ( m[medio] > n )
				de = medio - 1;
			else
				return ( medio + 1 );
		}

	}

	return ( - 1 );
}
```

Métodos de ordenación:

Se tratan de métodos encargados de ordenar una lista homogénea de elementos de manera ascendente o descendente.

Nos encontramos con tres tipos básicos, aunque aparecen mezclas de ellos:

- Intercambio
- Selección
- Inserción

La eficiencia de un método, se mide en el tiempo medio de ordenación, ya que algunos métodos son más eficientes que otros en ciertas circustancias (listas cortas de elementos, listas ordenadas en parte...) y en otras no. Por citar un ejemplo, la burbuja es uno de los métodos más eficientes cuando se trata de una lista corta de elementos, y así mismo es uno de los peores cuando las listas son largas, es por esto por lo qué mediremos su eficiencia siempre por el tiempo medio.

Ejemplos de métodos:

Intercambio: Burbuja, Quicksort.
Selección: Selección de mínimos.
Ordenación: Shell.

Implementación:

Ordenación mediante método burbuja:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <conio.h>

#define MAX 10

int dato[MAX];

void Burbuja( void );

void main( void )
{
      int i;
      clock_t inicio, fin;

      clrscr();

      /* Fijamos la semilla inicial */
      srand( ( unsigned ) time( NULL ) );

      /* Cargamos el vector de aleatorios 0 - 99 */
      for ( i = 0 ; i < MAX ; i++ )
            Dato[i] = ( rand() % 100 );

      /* Mostramos el vector */
      for ( i = 0 ; i < MAX ; i++ )
            printf( " %2d ", Dato[i] );

      inicio = clock();
      burbuja();
      fin = clock();

      /* Mostramos el tiempo invertido (en s.) */
      printf( "%f\n", ( fin - inicio ) / CLK_TCK );

      /* Y mostramos el vector ordenado */
      for ( i = 0 ; i < MAX ; i++ )
            printf( " %2d ", Dato[i] );
}
```

```
void Burbuja( void )
{
      int i, j;
      int v;

      /* Para cada posición buscamos el elemento más pequeño de la lista por su derecha */
      for ( i = 0 ; i < MAX ; i++ )
            for ( j = i + 1 ; j < MAX ; j++ )
            {
                  if ( Dato[i] < Dato[i] )
                  {
                        v = Dato[i];
                        Dato[i] = Dato[j];
                        Dato[j] = v;
                  }
            }
}
```

Notas:

Clock devuelve el número de ciclos de reloj desde que se encendió el ordenador. (18-19/s.).

CLK_TCK es una constante que nos indica el número de ciclos de reloj que hay por segundo.

Función burbuja genérica:

Ahora pretendemos conseguir una mayor abstracción y portabilidad de nuestra función de ordenación por el método burbuja:

Prototipo:

```
void Burbuja( void *item, int n, int  ( *cmp )( void *, void * ) );
```

hay que tener en cuenta que no podemos hacer item++ para avanzar (ya que void significa vacío no se sabe el tamaño del item).

Debemos emplear un tipo de dato que sea del mismo tamaño en todos los sistemas de hardware, en C el único dato con estas características es el char que siempre ocupa 1 byte.

Ahora necesitamos pasar un argumento más a la función: el tamaño en bytes del tipo de dato a ordenar.

```
void Burbuja( void *item, int n, int ( *cmp ) ( void *, void * ), unsigned int s )
{
	char *ptr = ( char * ) item;
	char c;
	int i, j, t;

	for ( i = 0 ; i < n ; i++ )
	{
		for ( j = i + 1 ; j < n ; j++ )
		{
			if ( ( *cmp ) (  ( void * ) &ptr[i * s], ( void * ) && ptr[j *
			s] ) > 0 )
			{
				for ( t = 0 ; t < s ; t++ )
				{
					i = ptr[i * s + t];
					ptr[i * s + t] = ptr[j * s + t];
					ptr[j * s + t] = c;
				}
			}
		}
	}

}
```

Notas:

Debido a que a priori no sabemos el número de bytes que formará cada elemento, lo trasladamos byte a byte cuando realizamos el intercambio.

La rutina de comparación variará de un tipo de datos a otro, pero nos bastará saber que hemos de devolver TRUE cuando el primer elemento sea mayor y FALSE en el resto de los casos.

Rutina de comparación de enteros:

```
int cmp_int( void *dato1, void *dato2 )
{
	if ( ( ( int ) dato1 ) > ( ( itn ) dato2 ) )
		return( 1 );
	else
		return( 0 );
}
```